An APL compiler

Timothy A. Budd

TR 81-17

Department

of

Computer Science

The University of Arizona

An APL compiler

Timothy A. Budd

TR 81-17

ABSTRACT

Almost all implementations of APL to date have been based on interpreters, rather than compilers. Even commercial systems described as APL "compilers" produce code at run time, and may produce code several times for the same APL statement during repeated executions. This paper describes a true compiler for APL. Furthermore the code produced by this compiler is *space-efficient*, in that large intermediate values are avoided by computing expressions element by element.

October 25, 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

An APL compiler

1. Introduction

Almost all implementations of APL to date have been based on interpretation, rather than compilation. This interpretative nature has some advantages; it contributes to a flexibility of use that facilitates program design and development, so that the productivity of programmers using APL is frequently much higher than that of programmers using other languages. Because APL exploits the use of arrays, the statement decoding overhead involved in this interpretive process is often only a small fraction of the total execution cost. Nevertheless, there are applications where, once a program has been developed, complied code may be advantageous in production programs.

There is a second problem with most APL implementations that is actually more bothersome in practice than the cost of interpretation. This is the fact that naive implementations of APL frequently generate large intermediate values on the way to producing small or even scalar results. The style of programming preferred by most experienced APL programmers (using arrays and parallelism as much as possible) tends to intensify this problem, and hence most APL programmers feel the space problem is typically more critical than the time problem.

This paper describes an implementation of a compiler which addresses both these problems. The system accepts a sequence of APL function definitions as input, and produces a sequence of subprograms in a high-level language (in this case C [7]) as output. The C programs are then passed to the C compiler which produces the final executable program.

The notion of compiling APL is not new. The first steps towards an APL compiler were taken by Abrams in his thesis [1]. Although no system was actually built, Abrams laid the foundation for almost all later work in this area.

Jenkins describes a system for translating a very restricted subset of APL into Algol [6]. This system imposed several language restrictions and required declarations in order that all rank, type and syntactic meanings could be known to the translator. By contrast, in the current system the only declarations required are those needed to disambiguate variables from labels and function references.

Perlis [11] discusses an alternative space-efficient method that he calls *ladders* and which, it is claimed, could lead to the development of an APL compiler. Using ladders an APL statement is implemented as a sequence of interacting co-routines. The ladder idea is similar to the concept of "boxes" used in the APL compiler, in that templates for each operand are created and then spliced together. However the code produced by the APL compiler is much simplier and more general than that produced using ladders. Miller [8] describes the ladder idea in more detail, however no compiler has actually been constructed.

Hardware changes to facilitate APL processing have been proposed, including producing special micro-programs to directly execute some APL operators [15], and designing special machines [9]. This approach, however, limits the use of the language to those possessing the special hardware, and more importantly prevents the interleaving of operations required for the space saving techniques described in Section 2.

The goals of the compiler discussed here are also similar to the system designed by Guibas and Wyatt [4]. They use the term *accessor* for what is here described as a *request vector*. Similarly they also make several code generation passes over the parse tree. However there are many significant differences at the design level. In particular the Guibas and Wyatt system is a "dynamic compiler" which produces code at run time (see discussion below). Furthermore they describe a method for producing code for only a small subset of APL operators, and do not extend their idea to the general case.

There exist several commercial "compilers" for APL, for example the Burroughs APL\7000 system or the Hewlett-Packard APL\3000 system. Instead of compiling an entire file, or even a function, these systems compile code for an individual statement the first time the statement is executed. This code is tied strongly to the types, ranks, and shapes of the objects manipulated by the statement. Along with the code for the statement a "signature" code sequence is produced which verifies these attributes. On subsequent executions this signature code is first evaluated. If it fails the entire statement is then *recompiled*. Thus the entire compiler must be kept as part of the run-time system. In the system described here the APL source is compiled *once*, in its entirety, before execution begins. Thus the need for a large run-time system is reduced.

2. Boxes

The code produced by the APL compiler is based on the idea of *boxes*, a notion first described by Richard Lipton. Consider a parse tree for an expression where each node represents a separate automaton, or box. Arcs in the parse tree correspond to communication lines, and along these lines can flow a number of *commands* or *requests*. For example Figure 1 shows the box structure which corresponds to the APL expression

A ← +/ 2 = + ⌿ 0 = (⍳ 200) ∘.| ⍳ 200

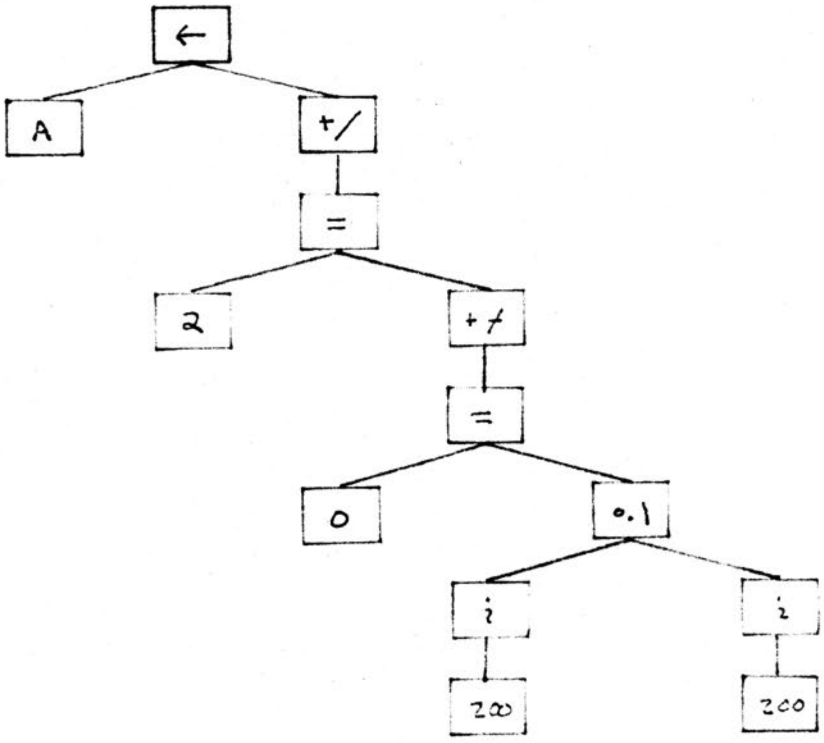

Figure 1: A Box Structure

This expression computes the number of primes less than 200. The term ⍳ 200 produces a vector of numbers running from 1 to 200. The operator | is modular division, hence the outer product (⍳ 200) ∘.| ⍳ 200 produces a matrix where the i,j element is i mod j. Comparing this matrix to zero produces a boolean matrix where the i,j element is 1 if and only if i mod j equals zero. The plus reduction of this matrix results in a vector of 200 elements, where position i in this vector contains the number of divisors of i. If this value is 2 (that is, the only divisors are 1 and the value itself) then the number is prime. Thus comparing the vector to 2 produces a vector with ones in the positions corresponding to primes. Taking the plus reduction of this vector gives the result, the number of primes less than 200.

To see how this result could be obtained using boxes, imagine the topmost box (the assignment automaton) has been signaled to begin. It first transmits the command "tell me what your result shape will be" to its child, the reduction box. The reduction box then transmits this requests to *its* child, which in turn passes it on again. Eventually the command is propagated throughout the parse tree, at which point answers start to be passed back up. The iota operators reply that they will produce a vector of 200 elements, a fact which the outer product box uses to determine that it will produce a 200 by 200 matrix. The equality operator box merely checks shape conformability for its operators, and does not alter the result shape. The reduction box calculates, however, that given a 200 by 200 matrix for input it will produce a 200 element vector. Again the equality automaton will leave this shape unaltered, but the topmost reduction box will, given a 200 element vector, produce a scalar result.

The assignment automaton thus knows that it will be receiving a scalar value. It therefore allocates enough storage to hold this value, and sends the command "produce your value" to the reduction box.

In order to compute this value, the reduction box needs to take the sum of the 200 element vector produced by the equality box. It therefore creates a variable, i, which loops from 1 to 200, and passes the sequence of requests "produce your i^{th} value" on to the equality operator. As it receives answers, it keeps a running sum, and only when all numbers have been added does it return its own value.

The second reduction box, given a request for its i^{th} value, creates a second variable, j, which loops from 1 to 200. It then passes on the request "produce your i,j value". Given this request, the outer product operator passes a request for its i^{th} value to its left child, and a request for is j^{th} value to its right child.

Once requests for values have reached to bottom of the parse tree, results start to propagate up. The reduction operators continue to loop, making requests on their children, until they have computed their own values. Eventually the single scalar result is passed back to the assignment box, which stores the value in the appropriate place and terminates.

The important fact to note about this process is that the only storage needed is to hold requests and results. The 200 element vectors computed by the iota operators and the 40000 element matrix computed by the outer product are never actually constructed, but are built up piece by piece as needed.

It is unlikely that a production system that operated in a manner similar to that just described could be very efficient. The costs involved in message passing and decoding would be prohibitive. Instead, in the APL compiler boxes refer to code generation routines, and the commands request the production of code fragments. These code fragments are spliced together by the compiler, and result in a very efficient run-time implementation.

In the current implementation there are four commands, three of which result in the generation of code fragments and a fourth used to pass information through the tree. These commands are as follows:

INFO Declare whatever compile-time information you may know (such as your ultimate result type and rank). Also declare the number of temporary locations you will need to evaluate your expression (since storage for arrays is allocated dynamically, only a single pointer is needed to declare storage for an array, which is important since result ranks and shapes will not, in general, be known at this point).

SHAPE Produce code which will, at run time, result in your type, rank, and shape information being placed into known locations. Also produce code to allocate whatever temporary locations you may require.

VALUE Given the name of a variable which will, at run time, contain a vector of values representing a fixed subscript position, produce code that will compute the value at that position, and place the result into a known location.

FINISH Produce code that will free whatever temporary storage you have allocated.

Within each box the order of incoming commands is always (INFO, SHAPE, VALUE, FINISH). However, in the system as a whole the sequence of commands is more complex, since some operators (such as iota or reshape) must evaluate one or both of their arguments before they can determine their resultant shape. This fact means that the order of execution is not strictly left to right, and this leads to some deviations from the language standard (see Section 3).

Using this space-efficient philosophy, there are some tradeoffs that can be made between temporary storage space and execution speed. Consider the outer product A ∘.+ B. Given a request for a specific element from the outer product, the space-efficient method is to break this request into separate requests for A and B, compute these values, and return their sum. If A and B are easily computed, this is the most efficient method.

If either A or B is a complex expression, however, this can be quite costly, since each element of B is computed once for each element of A, and vice versa. In this case it would be better to collect the values of A and B in temporary variables, and then to produce the outer product from these variables. Currently only the space-efficient method is used. One can envision, however, a system which uses some static information (the depth of the subexpression parse tree, for example) to make a choice at compile time between different implementation strategies. Further optimizations that are currently being considered are described in Section 7.

Space Efficient

- Scalar Operators (plus, times, etc.)
- Take Drop
- Monadic Reversal and Transpose
- Inner Product
- Iota
- Subscripting
- Encode Decode
- Shape
- Reduction

Time/Space Tradeoff

- Outer Product
- Scan
- Membership
- Index Of (Dyadic Iota)

Partial (Must evaluate one argument)

- Dyadic Reversal and Transpose
- Reshape
- Compress and Expand

Space Inefficient

- Roll and Deal
- Grade Up and Grade Down
- Function Calls
- Input Quad

Figure 2: Space Efficient Characterization of APL Operators

Certain operators, for example sorting or function calls, are not amenable to this sort of space-efficient handling. These are implemented by collecting their arguments, performing the operation in a conventional fashion, and returning their values, as requested, from the temporary variables produced. Figure 2 divides the APL operators into four groups. The first are those which require no intermediate storage, and thus have a very space efficient implementation. Operators in the second group also have a space efficient implementation, but this may result in slower execution speeds than a partially or fully space inefficient implementation. The third group lists operators which require values from one of their operands, but having these values then have a space efficient implementation for the remainder of the computation. The final group are those operators which are totally space inefficient, and which must be implemented by brute force.

3. Language Changes and Restrictions

There are several differences between the language accepted by this APL compiler and the language accepted by most conventional implementations (such as APLSV or APL\3000), or the language standard described in [2]. A few changes are required to remove ambiguities from the language, since the meaning of every token must be known at compile time. A second set of changes are made to improve the quality of the compiled code, for example to take advantage of type information supplied by the user. Finally a number of changes were necessitated by the compiled nature of the programs, as opposed to the interpretive workspace concept of systems like APLSV.

The "most recently defined" scope rule of APL [2] has been replaced by a two-level scoping rule, as in C [7]. Every variable is either local, in which case only the current function can reference it, or global, in which case any function can reference it. This change allows the compiler to produce simplier and much more efficient code. Experience indicates that the users seldom rely upon the conventional scoping rule used by APL. The conventional scoping rule could be implemented, but the system would have to keep more symbol table information around at run time.

The most obvious change is in the form of headings and the introduction of declarations. Function headings only define the name of the function, the name of the result variable, and the name of the arguments. A heading is followed by a sequence of declarations. In the initial implementation, all variables and all functions referenced must be declared. A variable is declared by the keyword **var** followed by a list of names, as in:

var a, b, c

In order to unambiguously recognize names, global variables and functions must also be declared, as in:

global d, e
fun x, y, z

Variables can take on any datatype (boolean, integer, real, or character). In general, more efficient code can be produced if the type of objects is known. Some type information (for example that the iota operator produces integers) can be inferred from the program. In addition, the user can optionally specify the types of variables or functions, as in the following:

int p
char fun r
real global q

Niladic functions are not supported. This is due to the inherent ambiguity in expressions such as "A − 7" where A is known to be a function. Here we can interpret "− 7" as an argument to the monadic function A, or "−" as a dyadic operator with arguments "7" and "A". This problem could be avoided by requiring (at least in ambiguous cases) valence declarations, but experience indicates that niladic functions are seldom used.

The execute operator is not supported, since the compiler is no longer present when the code is executed. Similarly only constants (or constant vectors) can be entered in response to quad input. Again both these restrictions could be relaxed at the expense of a considerable increase in the size of the run-time support.

As was mentioned in Section 2, the order of execution may not be strictly left to right. Expressions using side effects that depend on the order may produce different responses or may not work at all in the compiler system. Consider, for example, the expression

$$(A \leftarrow \iota 5) - A$$

Here the minus operator will ask for the shape of A, and the shape of the left hand expression. But in the course of computing the shape of the parenthesized expression the value, and potentially the shape, of A is altered. Thus the minus operator produces requests for A on the basis of obsolete information, and unpredicatable results ensue. (It is tempting to suggest that a solution would be to split the parse tree into two trees, the first dealing with the assignment and the second with the remainder of the expression. This, however, rules out some optimizations that can be made in the process of multiple assignment, such as x ← y

←z←3 3 ρι0, and can in fact produce wrong answers in the case of subscripted assignment.)

Since the workspace concept of APLSV has been discarded, most of the system functions and commands relating the workspace manipulation have been eliminated.

A more complete description of the language is given in Appendix 1.

4. Code Generation

In order to understand how a representative code generation box operates, it is first necessary to know a little about what the generated code looks like. The details of parsing the APL source into the internal tree representation are straightforward and are not presented here. The parse tree is represented by a sequence of linked nodes, where the structure of each node is as shown in Figure 3. The fields *left*, *right* and *axis* point to left, right and axis children (an axis can be an arbitrary expression). The field *nodetype* contains the node type (reduction, dyadic scalar, for example), while *optype* contains the operator type (plus, times). The field *rank* contains the rank of the expression (if known), *rtype* contains the result type (again if known), and *info* various information bits. The fields *trs*, *rqv*, *mp*, *res* contain integers which represent the names of various local temporary variables (see below). The fields *ptr1*, *ptr2*, and *ptr3* are used for different purposes by the different code generation boxes.

```
struct node {
   int nodetype;
   struct node *right, *left, *axis;
   int optype;
   int rtype;
   int rank;
   int info;
   int trs;
   int rqv;
   int mp;
   int res;
   int ptr1;
   int ptr2;
   int ptr3;
};
```

Figure 3: Parse Tree Node Definition

The generated code manipulates a number of different data structures. *Memory pointers* are a union type that can be used to point to any type of memory object (scalar or array). The basic unit of shape information is a *type-rank-shape* structure. Results are returned in *result structures*, which, like memory pointers, can refer to any type of scalar object. The C definitions for these three types of structures are shown in Figure 4. A fourth important structure is the *request vector*, which is simply an integer array, containing subscript information.

Local variables are created for whatever type-rank-shape, memory pointers, result structures or request vectors are needed during the **information** phase.

Each code generation box is unique, and hence none can be called "typical" of all boxes. However the monadic transpose box (Figure 5) can be used to illustrate some of the concepts involved in code generation. The box is called with three arguments; the parse tree node for the transpose, an integer **state**, which represents one of the four commands described in Section 2, and a logical variable which is true if the current node is the top of the parse tree for the statement (this last argument is important only in the boxes for assignment, assignment to quad, and function calls, since these are the only operators that can appear at the top of an expression)[1].

1. An implicit ⎕← is placed in front of all other statements.

```
union _mp {   /* memory pointers */
   int *ip;
   double *rp;
   char *cp;
};

struct _trs {  /* type rank shape value structures */
   int type;
   int rank;
   int *shape;
   struct _mp value;
};

union _res {  /* results */
   int i;
   double r;
   char c;
};
```

Figure 4: Run-time Structure Definitions

Dobox is a "switchboard", that interfaces boxes. Thus in the **information** phase the monadic transpose box (Figure 5) performs the following operations:

1 It allocates a type-rank-shape variable it will use in later phases. (all operators create their own variable for type-rank-shape information).

2 It allocates a request vector pointer, passing the name down to its child.

3 It calls its child node in mode **info**, so that the child subtree can allocate whatever variables it needs (and so that type information can be propagated back up the tree).

4 It defines the result type returned for this variable to be the result type returned by the child node.

Note that no code is produced in the **info** phase. The **shape** phase determines the shape of the child subtree. When called in **shape** mode, the monadic operator box immediately calls its child. It is assumed that following this call code has been produced which places the shape information for the child in the child's trs variable (indicated by (node->right)->trs). The routines cptype and cprank are simple code generation templates, shown in Figure 6. Alcshape produces code which will allocate enough space for a shape vector. Notice the assumption that the trs structure contains the proper rank information at *run* time, and that it is not required that rank information be known at *compile* time. Alcrft produces code which will allocate a request vector of the appropriate size. A simple loop is then created which reverses the shape vector of the right child, placing the information into the shape vector of the current node. Again, the code for this loop is the same regardless of the object ranks.

Values are requested in the **value** phase by placing indices into a request vector. For example, if the result of an expression is a three dimensional array, the request vector contains three elements. Given a request vector (assumed to have been allocated and filled by some superior node in the parse tree), the monadic transpose box creates a new request vector (the space for which was allocated in the shape phase previously) which contains the indices in reverse order. The value of the child is then requested. Again the code produced to accomplish this is independent of the rank of the data.

In the **final** phase code is produced which will free the space allocated in the shape phase for the shape vector and the request vector. Again freshape and frerqv are simple code template routines, given in Figure 6.

```
/* bxtrans - monadic transpose */
bxtrans(node, state, top)
  struct node *node;
  int state, top;
{
  switch(state) {
default: caserr("bxtrans", state);

case INFO:
      node->trs = alctrs();
      (node->right)->rqv = alcrqv();
      dobox(node->right, INFO, 0);
      node->res = (node->right)->res;
      node->rtype = (node->right)->rtype;
      node->rank = (node->right)->rank;
      break;

case SHAPE:
      dobox(node->right, SHAPE, 0);
      cptype(node->trs, (node->right)->trs);
      cprank(node->trs, (node->right)->trs);
      alcshape(node->trs, node->trs);
      alcrft(node->rqv, node->trs);
      rankloop(node->trs);
      printf("_trs%d.shape[_i0] = _trs%d.shape[(_trs%d.rank - _i0) - 1];\n",
                  node->trs, (node->right)->trs, node->trs);
      break;

case VALUE:
      rankloop(node->trs);
      printf("_rqv%d[_i0] = _rqv%d[(_trs%d.rank - _i0) - 1];\n",
                  (node->right)->rqv, node->rqv, node->trs);
      dobox(node->right, VALUE, 0);
      break;

case FINISH:
      freshape(node->trs);
      frerqv(node->rqv);
      dobox(node->right, FINISH, 0);
      }
}
```

Figure 5: The Monadic Transpose Code Generation Box

5. An Example

Figures 7 and 8 show a partial listing of the code produced for the prime number producing expression

⎕ ← (2 = +/ 0 = (ι 200) ∘. | ι 200)) / ι 200

Corresponding to the three passes over the parse tree, the compiled code divides naturally into three fairly distinct components. The first group is a lengthy block of more or less straight line code that computes the type, rank and shape information for each operator in the expression. (The phrase "more or less" indicates

```
cptype(t1, t2) int t1, t2;{
  printf("_trs%d.type = _trs%d.type;\n",t1,t2);
}

cprank(t1, t2) int t1, t2;{
  printf("_trs%d.rank = _trs%d.rank;\n",t1,t2);
}

alcshape(t1, t2) int t1, t2;{
  printf("_trs%d.shape = _alcmem(_trs%d.rank, INT);\n", t1, t2);
}

alcrft(r, t) int r, t;{
  printf("_rqv%d = _alcmem(_trs%d.rank, INT);\n", r, t);
}

rankloop(t) int t;{
  printf("for (_i0 = 0; _i0 < _trs%d.rank; _i0++)\n", t);
}

freshape(t) int t;{
  printf("_memfree(_trs%d.shape);\n",t);
}

frerqv(r) int r;{
  printf("_memfree(_rqv%d);\n",r);
}
```

Figure 6

that this statement is not exact. Several operators, such as transpose, catenate, or inner product, actually produce loops or have multiple cases, but generally these loops are small in comparison to the computation loops. In addition, some operators, such as compress or reshape, may have to compute subtree values in order to determine their own shapes).

The second block of code computes the actual result values for the expression. Generally this involves one or more nested loops. Although the size of this code block is typically smaller than the first, because of the loops it is responsible for a much larger part of the actual running time of the statement. The final group is the smallest of the three, and usually consists of a few calls on the memory management routine **memfree**.

Currently all operators allocate and fill their own type-rank-shape structure. Since type and rank information is also propagated where possible at compile time much of the information stored in these local variables is never used. In designing the first version of the compiler it was argued that having a uniform structure for code generation boxes would make the compiler less complicated, and the inefficiencies involved would actually make little difference in the eventual run time speed, since the rank and shape computations were for the most part straight line code. Experience with the compiler indicates, however, that statements containing several APL operators can generate tens or even hundreds of C statements just to compute type, rank and shape information. Clearly this can become a sizable burden, particularly if, as Saal indicates [13] most of this information could be inferred at compile time.

On the other hand, attempting to propagate constants (type and rank information) and eliminate dead code would probably result in at least a 25% increase in the size of the compiler. An alternative would be to pass the resulting C code through an optimizing compiler [14] and let it clean things up.

```
/* compute trs for constant 2 */
    _trs2.type = INT;
    _trs2.rank = 0;
    _trs2.shape = &l_main[1];
/* compute trs for constant 0 */
    _trs5.type = BIT;
    _trs5.rank = 0;
    _trs5.shape = &l_main[1];
/* compute trs for iota 200 */
    _trs7.type = INT;
    _trs7.rank = 1;
    _trs7.shape = &l_main[6];
/* compute trs for iota 200 */
    _trs8.type = INT;
    _trs8.rank = 1;
    _trs8.shape = &l_main[7];
/* compute trs for outer product */
    _trs6.rank = _trs7.rank + _trs8.rank;
    _trs6.shape = _alcmem(_trs6.rank, INT);
    _catvec(_trs6.shape,_trs7.shape,_trs7.rank,_trs8.shape,_trs8.rank);
    _trs6.type = _optype(ABS, _trs7.type, _trs8.type);
/* compute trs for equality operator (0 = expression */
    _scatrs(&_trs5, &_trs6, &_trs4);
    _trs4.type = _optype(EQ, _trs5.type, _trs6.type);
/*  compute trs for plus reduction */
    _trs3.type = INT;
    _trs3.rank = (_trs4.rank > 0) ? _trs4.rank - 1 : 0;
    _i1 = 0;
    _trs3.shape = _alcmem(_trs3.rank, INT);
    if (_trs3.rank > 0)
      _cpbut(_trs3.shape, _trs4.shape, _trs4.rank, _i1);
    else
      _trs3.shape = &l_main[1];
    _rqv3 = _alcmem(_trs4.rank, INT);
/* compute trs for equality operator (2 = expression) */
    _scatrs(&_trs2, &_trs3, &_trs1);
    _trs1.type = _optype(EQ, _trs2.type, _trs3.type);
/* compute trs for iota (right hand side of compress) */
    _trs9.type = 2;
    _trs9.rank = 1;
    _trs9.shape = &l_main[8];
/* compute trs for compression */
    _i2 = (_trs9.rank > 0) ? _trs9.rank - 1 : 0;
    if (_trs1.rank != 0)
      if (*_trs1.shape != _trs9.shape[_i2])
        error("conformability error");
    _trs10.type = _trs9.type;
    _trs10.rank = (_trs9.rank > 0) ? _trs9.rank : 1;
    _trs10.shape = _alcmem(_trs10.rank, INT);
    *_trs10.shape = *_trs9.shape;
```

Figure 7: Type-Rank-Shape Information Gathering for Prime Example

```
/* allocate an area for compression indices */
    _trs10.value.ip = _alcmem(_trs9.shape[_i2], INT);
    _mp2.ip = _trs10.value.ip;
    _rqv4 = _alcmem(1, INT);
/* loop gathering information for compression */
    for (*_rqv4 = 0; *_rqv4 < _trs9.shape[_i2]; (*_rqv4)++) {
      _res1.i = 2;
      _cpadd(_rqv3, _rqv4, _trs3.rank, _i1);
      _res2.i = 0;
      _mp1.ip = &_rqv3[_i1];
/* loop doing plus reduction */
      for (*_mp1.ip = _trs4.shape[_i1] - 1; *_mp1.ip >= 0; (*_mp1.ip)--) {
        _res3.i = 0;                /* constant zero    */
        _res4.i = _io + *_rqv3;      /* left iota        */
        _rqv2 = &_rqv3[1];
        _res5.i = _io + *_rqv2;      /* right iota       */
        _res5.i %= _res4.i;         /* modular division */
        _res3.i = (_res3.i == _res5.i);/* test 0 = result  */
        _res2.i += _res3.i;         /* keep running sum */
      }
      _res1.i = (_res1.i == _res2.i);  /* test 2 = result  */
      if (_res1.i == 1)             /* if true          */
        *(_mp2.ip++) = *_rqv4;       /*  keep index      */
    }
/* free storage */
    if (_trs3.rank > 0)
      _memfree(_trs3.shape);
    _memfree(_rqv3);
    _memfree(_trs6.shape);
/* compute shape of compression */
    _trs10.shape[_i2] = (_mp2.ip - _trs10.value.ip);
    _memfree(_rqv4);
    _rqv4 = _alcmem(_trs10.rank, INT);
    _rqv1 = _alcmem(_trs10.rank, INT);
    _cpp = 0;
/* loop built by output operator */
    if (_numin(&_trs10) > 0) {
      for (_i0 = 0; _i0 < _trs10.rank; _i0++)
        _rqv1[_i0] = 0;
      do {
        *_rqv4 = _trs10.value.ip[*_rqv1];
        _res6.i = _io + *_rqv4;      /* iota */
        _prntres(stdout, _res6, _trs10.type);
      }
      while (_pbmprqv(&_trs10,_rqv1, stdout, 1));
      _memfree(_rqv1);
    }
    _memfree(_rqv4);
    _memfree(_trs10.shape);
    _memfree(_trs10.value.ip);
```

Figure 8: Computation Code Produced for Prime Example

As noted previously, the compress operator requires knowledge of the left operand values before it can determine its shape. The first loop in Figure 8 is therefore designed to compute a vector, _trs10.value.ip, containing the positions of the left operand that contain ones. By recognizing that the iota operator produces a vector, several optimizations have been made possible. For example the loop for the reduction operator and the loop for the membership operator use pointers, rather than less efficient but more general subscripted expressions. Following the first loop, the vector _trs10.value.ip is used to compute the shape of the compress operator.

The final loop is built by the output operator, and illustrates how loops are constructed when rank and shape information is unavailable at compile time. (Even though shape information is known in this case, the execution of output loops is dominated by I/O time, and hence are not optimized). the routine **numin** determines the number of elements in the compress result. If this number is nonzero, a request vector of the appropriate size is constructed. The compress operator (which, unlike the output box, recognizes that it is dealing with a vector) then transforms this into a new request vector, _rqv4. This is passed to the iota operator, which produces the final answer. **pbmprqv** then increments the request vector _rqv1, speedometer fashion, also printing a *newline* at the end of each row. **pbmprqv** returns false (zero) when all elements have been printed.

6. Execution Timings

In order to compare the execution speed of the code produced by the APL compiler to that of other systems, a test harness was written, as shown in Figure 9. The statement labeled L is executed one thousand times. A baseline was established by timing the program using an empty statement for L. This baseline was subtracted for all subsequent timings, to factor out the overhead and initialization costs of running the harness.

```
bit vl
int vi, mi
real mr, vr
char vc, vc
vl ← 1 = 1 0 1 1 0 0 0 1
mi ← 10 10 ρ vi ← (500 ρ 0 1 0 0 1)/ι 500
mr ← 10 10 ρ vr ← vi + .1
mc ← 26 26 ρ vc ← 'abcdefghijklmnopqrstuvwxyz'
i ← 0
L: statement
-> (1000 > i ← i + 1) / L
```

Figure 9: Program Used to Obtain Execution Timings

Timings for several statements are given in Table 10. These are compared with similar timings for various APL implementations taken from a study by Harris Computer Systems [5]. Interpreting the comparisons is made difficult, however, by the presence of two unknowns, the basic machine differences and differences in implementation methods. Nonetheless, there are several conclusions that can be drawn from these figures.

The first observation is that type declarations can have a significant impact on execution speed. The plus reduction of the integer vector vi takes 4.56 milliseconds. The plus reduction of the real vector vr, however, takes 12.98 milliseconds, almost 3 times longer. Furthermore if vr is declared to be **var** (type unknown) the plus reduction takes 54.28 milliseconds, a tenfold increase over the integer time. This large increase in execution time is caused by the presence of a number of subroutine calls in the innermost loops.

Some of the larger execution times can be explained by the choice of algorithms used to implement various operators. For example the UNIX quickersort utility was used to implement the gradeup and gradedown operators. This, however, requires one subroutine call for every comparison. It is likely that some execution

	APL Compiler	IBM 370 158	DEC 2040	HP 3000 Series II	CDC Cyber 73	BURROUGHS B7700
z ← +/ vi	4.56	.9	2.5	50.6	2.0	4.6
z ← +/ vr	12.98					
z ← ∨/ vl	1.2	.5	5.1	5.4	.5	.8
z ← ⌈/[1] mi	15.24	1.1	4.6	39.3	1.7	1.7
z ← | vr	13.18	3.5	6.2	101.7	1.8	1.6
z ← vr[vi[ι 20]]	3.28	1.1	4.3	34.4	4.5	3.2
z ← vi[⍋ vi]	94.1	14.7	56.0	288.1	40.4	32.8
z ← ¯2 1 ↑ mr	2.84	.9	2.2	5.1	.9	1.3
z ← vi ∊ vi	140.6	61.5	441.2	3871.0	263.6	17.4
z ← vr ⌊ . + vr	26.32	4.1	7.7	95.3	6.6	9.9

Table 10: Execution Timings for Several APL Statements in Milliseconds

time improvement could be realized by expanding the code for sort in line.

A dramatic example of the effect of different implementation strategies is provided by the membership (∊) operator. A ∊ B is defined to be a boolean expression of the same rank and shape as A, where a one indicates the corresponding element is present in B and a zero indicates absence. In an early implementation, in keeping with the space-efficient philosophy, when a value was requested the code first produced the corresponding element in A, and then looped through all elements of B looking for a match. If B was a complex expression, however, this repeated evaluation became quite costly. For example in one test program there occurred the statement

```
v ← x[+\(ι+/y)∊¯1↓1++\0,y]
```

By introducing a new variable z and changing this to

```
z ← ¯1↓1++\0,y
v ← x[+\(ι+/y)∊z]
```

execution time for the entire program was observed to drop from 34.2 to 2.0 seconds.

Clearly it would be possible for the compiler to create a temporary variable, collect the right hand operand, and at least in this case, produce faster code. Given this information, the implementor is faced with a difficult choice. Should the space-efficient philosophy be abandoned in the case of membership in order to improve execution time, or will this result in an inordinate waste of space? One compromise that can be envisioned would have been to define some measure of the average cost to access an element of an expression. This figure could easily be computed during the information gathering phase of code generation. If the cost of accessing the right-hand side of the membership operator became too large, the space-efficient philosophy could be abandoned.

However, an even greater time, if not space, efficient method was chosen. If we let N and M represent the number of elements in A and B, respectively, then the algorithm presented above requires, in the worst case, O(N×M) steps to evaluate A ∊ B. An alternative is to first collect and *sort* B, which can be done in O(M log(M)) steps. Binary search can then be used to determine if any particular element occurs in B, using only O(log(M)) steps. Thus in the worst case the complexity of evaluating A ∊ B is reduced to O((N+M) log(M)) steps. Using this method the execution time for the benchmark expression z ← vi ∊ vi was observed to drop from 334.2 to 140.6 milliseconds.

Ongoing empirical studies are being conducted to evaluate the effectiveness of these, and other, alternative implementation strategies.

7. Optimizations

There are several ways that the code produced by the APL compiler could be improved. The first approach is to improve the efficiency of the code generated without altering the actual algorithms used. For example, several operations that are currently performed by calling run-time subroutines could be expanded in line, thus saving the cost of a subroutine call. Similarly there are code sequences generated that use subscripts but which could be modified to use pointers, since pointers usually have a more efficient implementation.

A second method for optimizing computations is to improve algorithms. A characteristic feature of many APL operators, such as grade up or inner product, is that they describe what is to be accomplished without describing how it is to be done. As was illustrated in Section 6, the choice of different implementation strategies can have a dramatic impact on execution time. The current implementation has almost always used a straightforward, easy to implement algorithm, even though in many cases a more complicated but more efficient one could be written. There are also many special cases that could be recognized; for example the scan operator, like the membership operator discussed in Section 5, evaluates its argument several times. This can be avoided if the results are accessed in raval order and the operator is commutative.

One aspect of this latter approach would be the recognition of *idioms*. In APL an idiom is a frequently occurring pattern of operators. As a language, APL is rich in idioms [12]. Frequently the recognition of an idiom could lead to a more efficient implementation of some expression. For example the sorting idiom A[⍋A] could be recognized as a single operator, and not as the two separate operators (grade up and subscripting) used to represent it. Other idioms are more complex, but their recognition could lead to an even greater savings in terms of space and time.

8. Status

The compiler is written entirely in C, using the YACC compiler-compiler available on the UNIX[2] timesharing system. Implementing the compiler required approximately three man months of effort. The compiler itself is rather large, comprising over 7000 lines of C code.

Current research is underway to evaluate the cost and benefits of some of the optimizations described in the last section. It is expected that future versions of the compiler will include some or all of these features.

Acknowledgements

I want to thank Alan Perlis for introducing me to APL programming. The idea that an APL compiler could be implemented in this fashion was suggested to me by Richard Lipton. I also benefited from several discussions with Brownell Chalstrom on ways of implementing several APL operators. Chris Fraser, Ralph Griswold and Gene Myers provided detailed comments on earlier drafts of this paper.

2. UNIX is a trademark of Bell Telephone Laboratories

References

[1] P. Abrams, "An APL Machine", SLAC report #114, Stanford University, 1970

[2] A. Falkoff and D. Orth, "Development of an APL Standard", APL Quote Quad, 4(2), pages 409-453, June 1979

[3] L Gilman and A. Rose, *APL an Interactive Approach*, Wiley, 1976

[4] L. Guibas and D. Wyatt, "Compilation and Delayed Evaluation in APL", Proceedings of the 5th ACM Symposium on Principles of Programming Languages

[5] Harris Computer Systems, "APL Comparison Chart", May 1979

[6] M. Jenkins, "Translating APL - An Empirical Study", Proceedings of the APL 75 Conference, Pisa, Italy, 1975

[7] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, 1978

[8] T. Miller, *Tentative Compilation: A Design for an APL Compiler*, PhD Thesis, Yale University, 1978

[9] C. Minter, *A Machine Design for Efficient Implementation of APL*, PhD Thesis, Yale University, 1976

[10] R. Polivka and S. Pakin, *APL: The Language and Its Usage*, Prentice Hall, 1975

[11] A. Perlis, "Steps toward an APL Compiler - Updated", Yale University Department of Computer Science Research Report #24 (1975)

[12] A. Perlis and S. Rugaber, "The APL Idiom List", Yale University Department of Computer Science Research Report #87 (1977)

[13] H. Saal and Z. Weis, "Some Properties of APL Programs", Proceedings of the APL 75 Conference, Pisa, Italy, 1975

[14] W. Wulf, R. Johnsson, C. Weinstock, S Hobbs and C. Geschke, *The Design of an Optimising Compiler*, American Elsevier, 1975

[15] R. Zaks, *A Microprogrammed APL Implementation*, Sybex, Berkeley, Calif. 1978

Appendix 1
Language Changes

Except as noted in this appendix, the APL statement syntax has been left unchanged. Detailed descriptions of this syntax can be found in several textbooks, for example *APL: The Languages and Its Usage*, by Polivka and Pakin [10], or *APL An Interactive Approach*, by Gilman and Rose [3].

1. Workspaces

The concept of the APL workspace is not supported. Workspace commands are not recognized and result in syntax errors if used. Built in functions, such as ⎕WA or ⎕LC, that refer to workspace parameters are not recognized.

2. Scoping Rules

The APL "most recently used" scoping rule has been replaced by a simple two-level variable scoping. Variables are either local to the program in which they are declared, or global to all procedures. Variables declared outside of any program are automatically given the attribute global (see Section 5).

System variables, such as ⎕IO and ⎕PP, are true global variables, and are not automatically restored to their previous values on procedure exit.

3. Order of Execution

The order of execution is not guaranteed to be strictly right to left. Certain operators, such as reshape or compress, may evaluate their left argument before their right argument. Thus expressions which depend upon a side effect may produce unpredictable results. An example is using a variable while at the same time redefining the variable elsewhere in the expression. The following statements are almost certain to produce a result other than that intended.

```
X ← 3 3 ρ ι 9
(X ← ι 5) + X
```

4. Heading and Declarations

The format for procedure headings has been altered considerably. Local variables are no longer declared on the same line as the procedure heading. Instead, a procedure heading can be followed by any number of declaration statements.

The format for a heading is the symbol △, followed by an optional assignment part, followed by an optional left argument name, followed by the function name, followed by the right argument name. Niladic functions are not supported.

The syntax for a declaration is an attribute followed by a list of variable names. Attributes are divided into classes (**global**, **function**) and types (**var**, **int**, **bit**, **char**, **real**). An attribute consists of a class and/or a type. Thus the following are legal declarations and have the indicated meanings:

var a, b, c	local variables, type unknown
global int i, j	global integer variables
char global x	global character variable
fun p	function p, type unknown

All global variables and functions used in a procedure must be declared. Local variables need not be declared if their first occurrence is as the left argument in an assignment, but from a stylistic point of view it is better to declare all variables. Variables and functions need not have declared types, however specifying types allows the compiler to produce better code.

System variables, such as ⎕IO and ⎕PP, cannot be declared local to a procedure.

5. The program MAIN

Statements, including declarations, that are outside the range of any procedure body are assumed to refer to the main program. These statements can be intermixed with procedure declarations, however from a stylistic point of view this should be avoided. All variables in the main program are given the attribute **global**.

6. Execute

The execute operator is not supported. Neither are the system functions ⎕CR and ⎕FX. See Section 10 for a complete list of system functions.

7. Tracing

The function tracing commands S∇FN and T∇FN are not recognized. Instead there is the compiler command *trace*. The syntax for the trace command is the keyword **trace** followed by an optional integer. Numeric values have the following meaning:

0	(default) Turn tracing off
1	Print statement numbers as statements are executed.
2	Print the resulting values as statements are executed.
3	Print type, rank and shape information for each operator as it is computed.

Tracing is enabled until a subsequent **trace** command is encountered. Note that trace is a compiler command, not a dynamic run time command.

8. Axis specifiers

In general axis specifiers can be arbitrary expressions. The single exception is with the catenate operator, since the compiled code for catenate differs from that for laminate. The choice of whether to compile code for catenate or laminate is made on the basis of the *type* of axis value: integer types compile as catenate and real types compile as laminate. Thus even if a real expression results in a value with zero fractional part, the operation is still a laminate.

9. System Variables

The following system variables can be referenced, but not assigned.

⎕TS	returns an integer vector containing the current year, month, day, hour, minute, and second.
⎕AV	returns an 256 element character vector representing the ASCII character sequence.

The following system variables can be both assigned to and referenced.

⎕IO	Sets the index origin for indexing. Returns the current index origin.

⎕PP Sets the number of positions used in printing integer and real variables. Returns the current printing precision.

⎕PW Sets the maximum width of the output line. Returns the current printing width.

⎕RL Sets a seed value for the random number generator used in roll and deal. Returns the current random number.

10. System Functions

System functions relating to input and output are described in the next section. Additionally, the following system functions are available.

⎕AG (monadic) takes an integer i and returns the ith argument string from the command line.

⎕EX (monadic) takes a string vector and executes it as a command in a UNIX subshell.

11. Input / Output

Facilities for opening and closing files are included. The open system function takes a string representing the file to be opened, and returns an integer, called the *file descriptor*, that is used in subsequent requests to read or write to or from the file. File input or output is indicated by the presence of a file descriptor as a left argument to quad or quadquote.

⎕OP (monadic) Takes a string vector and opens the indicated file for writing. Produces and error message if the file cannot be opened.

⎕OP (dyadic) The right argument is as above. The left argument is one of 'r', 'w' or 'a', and the file is opened for reading, writing or appending, respectively.

⎕CL (monadic) The argument is an integer returned by a previous call to ⎕OP. The associated file is closed.

The following example shows the string 'abc' being written to the file 'temp'.

```
i <- 'w' ⎕OP 'temp'
i ⍞ <- 'abc'
```

Appendix 2
TTY Menomiçs

The following tty menomics can be used in place of the indicated APL symbols. In some instances two or more symbols are given, in which case either form may be used.

|	absolute value	|
∧	logical and	&
←	assignment	_ <-
⎕	input or output quad	.bx
⌈	ceiling	.ce
○	circle	.lo
⍝	comment	"
/	compress	/
⊥	decode	.dc
∆	del	.dl
÷	division	%
↓	down arrow	.da
⊤	encode	.ec
∊	epsilon	.ep
=	equal	=
*	exponentiation	*
\	expansion	\
⍀	expansion along first coordinate	.fs
!	factorial	!
⌊	floor	.fl
⍕	format	.fm
→	goto	-> .go
⍒	grade down	.gd
⍋	grade up	.gu
>	greater than	>
≥	greater than or equal	>=
⍳	iota	.io
⍟	log	.lg
<	less than	<
≤	less than or equal	<=
⌹	matrix division	.md
−	minus	-
⍲	name	.nd
⍱	nor	.no
~	not	~
¯	negation	.ng
≠	not equal	!=
∨	or	.or
+	plus	+
⍞	quote quad	.qq
⌿	reduce along first coordinate	.bs
⌽	reversal	.rv
⊖	reversal along first coordinate	.cr
⍴	rho	.ro
?	roll	?
∘	small circle	.so
×	times	#
⍉	transpose	.tr
↑	up arrow	.ua

System variables and system functions are indicated by preceding the name with .bx, as in .bxav for ⎕AV.